Module 9: Lists

Thomas Schwarz, SJ
Marquette University, 2018
Lists

- Python is a high-level programming language with built-in sophisticated data structures
- The simplest of these data structures is the list.
- A list is just an ordered collection of other objects
  - The type of the objects is not restricted
- Let’s start unpacking this a bit.
Lists

- We create a list by using the square brackets.
  - `alist = [1, 3.5, "hello"]`
    - A list with three elements of three different types
  - `blist = [1, 3.5, "hello", 1]`
    - A list with four elements, where one element is repeated
  - `clist = [1, "hello", 3.5]`
    - A different list than `alist`, but with the same elements
- The order is different
Lists

• Accessing elements in a list

• We access elements in a list by using the square brackets and an index

• Indices start at 0

• Example:

• \texttt{lista} = [‘a’, ‘b’, ‘c’, ‘d’]

• \texttt{lista}[0] is ‘a’

• \texttt{lista}[1] is ‘b’

• \texttt{lista}[2] is ‘c’
Lists

• Python uses negative numbers in order to count from the back of the list

  • `lista = ['a', 'b', 'c', 'd']`
  
  • `lista[-1]` is the last object, namely the character ‘d’
  
  • `lista[-2]` is the second-last object, namely the character ‘c’
  
  • `lista[-4]` is the first object, namely the character ‘a’
Manipulating Lists

- We manipulate lists by calling list methods
- You should read up on lists in the Python documentations
  - https://docs.python.org/3/tutorial/datastructures.html
- The length (number of objects in a list) is obtained by the `len` function.

```python
>>> lista = [1, 2, 3]
>>> len(lista)
3
```
Manipulating Lists

• We add to a list by using the append method
  
  • Example:

```
>>> lista = [1, 2, 3]
>>> lista.append(5)
>>> lista.append([1,2])
>>> print(lista)
[1, 2, 3, 5, [1, 2]]
```

• The resulting list `lista` has five elements, the last one being a list by itself.

• The append method always adds an element at the end.
Manipulating Lists

- The opposite of `append` is `pop`.
  
  Whereas `append` returns the special object `None`, `pop` removes the last element in the list and returns it.

- Example
  ```python
  >>> lista = [1,2,3]
  >>> lista.pop()
  3
  >>> print(lista)
  [1, 2]
  ```
Manipulating Lists

- We can also combine two lists with extend.
  - The method parameter is a list that is added to the first list.
    ```python
    >>> list1 = [1, 2, 3]
    >>> list2 = [4, 5]
    >>> list1.extend(list2)
    >>> list1
    [1, 2, 3, 4, 5]
    ```
    - This is different than appending.
      ```python
      >>> list1 = [1, 2, 3]
      >>> list2 = [4, 5]
      >>> list1.append(list2)
      >>> print(list1)
      [1, 2, 3, [4, 5]]
      ```
    - The resulting list has four elements, with the last one being a list
Manipulating Lists

- To remove items from a list, we can use
  - remove
  - del

- The remove method removes the first element from the list that matches a parameter
  - It does not remove all elements

- Example:

  ```python
  >>> lista = [1, 2, 3, 4, 5, 1, 1, 2, 2, 2, 3]
  >>> lista.remove(1)
  >>> lista
  [2, 3, 4, 5, 1, 1, 2, 2, 2, 3]
  ```
Manipulating Lists

- del operator:
  - A generic operator
  - In order to remove an item from a list, you specify a list and an index
  - Example: Remove the third element (“c”) from a list

```python
>>> lista = ["a", "b", "c", "d", "e"]
>>> del lista[2]
>>> lista
["a", "b", "d", "e"]
```
Manipulating Lists: A Standard Pattern

• A pattern for list modification
  • Often, we need to process a list
    • A standard pattern:
      • Create an empty result list
      • Walk through the processed list
      • Add elements to the result list
Manipulating Lists: A Standard Pattern

• Example:

• Filtering:

  • Retain all elements in a list that are even numbers

```python
def even(lista):
    result = []
    for ele in lista:
        if ele%2==0:
            result.append(ele)
    return result
```

```
>>> even([1, 2, 3, 6, 7, 98, 12, 324, 43, 56, 15, 37, 45])
[2, 6, 98, 12, 324, 56]
```
Manipulating Lists: A Standard Pattern

Example:

Filtering:

Retain all elements in a list that are even numbers

def even(lista):
    result = []
    for ele in lista:
        if ele%2==0:
            result.append(ele)
    return result

>>> even([1,2,3,6,7,98,12,324,43,56,15,37,45])
[2, 6, 98, 12, 324, 56]
Manipulating Lists: A Standard Pattern

- Example:
  - Filtering:
    - Retain all elements in a list that are even numbers

```python
def even(lista):
    result = []
    for ele in lista:
        if ele%2==0:
            result.append(ele)
    return result
```

```python
>>> even([1,2,3,6,7,98,12,324,43,56,15,37,45])
[2, 6, 98, 12, 324, 56]
```
Manipulating Lists: A Standard Pattern

• Example:

• Filtering:

  • Retain all elements in a list that are even numbers

```python
def even(lista):
    result = []
    for ele in lista:
        if ele%2==0:
            result.append(ele)
    return result
```

```python
>>> even([1,2,3,6,7,98,12,324,43,56,15,37,45])
[2, 6, 98, 12, 324, 56]
```
Manipulating Lists: A Standard Pattern

- Example:
  
  - Filtering:
    
    - Retain all elements in a list that are even numbers

```python
def even(lista):
    result = []
    for ele in lista:
        if ele%2==0:
            result.append(ele)
    return result
```

```python
>>> even([1,2,3,6,7,98,12,324,43,56,15,37,45])
[2, 6, 98, 12, 324, 56]
```
Manipulating Lists: A Standard Pattern

- Example:
  - Map — transforming all elements in a list
    - Given a list of numbers, round them to the nearest digit after the decimal point
Manipulating Lists: A Standard Pattern

• Example:

def rounding(lista):
    result = []
    for ele in lista:
        result.append(round(ele,1))
    return result

>>> rounding([.113241, 123.45, 1342.68, 12, 123.456, 908.17, -89.1])
[0.1, 123.5, 1342.7, 12, 123.5, 908.2, -89.1]
Manipulating Lists: A Standard Pattern

• Example:

```python
def rounding(lista):
    result = []
    for ele in lista:
        result.append(round(ele,1))
    return result
```

```python
>>> rounding([.113241, 123.45, 1342.68, 12, 123.456, 908.17, -89.1])
[0.1, 123.5, 1342.7, 12, 123.5, 908.2, -89.1]
```
Manipulating Lists: A Standard Pattern

• Example:

```python
def rounding(lista):
    result = []
    for ele in lista:
        result.append(round(ele, 1))
    return result
```

```python
>>> rounding([.113241, 123.45, 1342.68, 12, 123.456, 908.17, -89.1])
[0.1, 123.5, 1342.7, 12, 123.5, 908.2, -89.1]
```
Manipulating Lists: A Standard Pattern

• Example:

```python
def rounding(lista):
    result = []
    for ele in lista:
        result.append(round(ele,1))
    return result
```

```python
>>> rounding([0.113241, 123.45, 1342.68, 12, 123.456, 908.17, -89.1])
[0.1, 123.5, 1342.7, 12, 123.5, 908.2, -89.1]
```
Manipulating Lists: A Standard Pattern

• Example:

```python
def rounding(lista):
    result = []
    for ele in lista:
        result.append(round(ele,1))
    return result
```

```
>>> rounding([.113241, 123.45, 1342.68, 12, 123.456, 908.17, -89.1])
[0.1, 123.5, 1342.7, 12, 123.5, 908.2, -89.1]
```
Manipulating Lists: A Standard Pattern

- We can generate this example to all functions of list elements

```python
def apply(function, lista):
    result = []
    for ele in lista:
        result.append(function(ele))
    return result
```

- This pattern is so important that Python 3 has a more elegant way of doing it. It is called list comprehension

- The apply function was part of Python 2, deprecated in Python 2.3 and abolished in Python 3.5
Lists are objects

- Lists are objects

- Objects have methods
  - Methods are functions that are called with an object as a parameter, but that are specific to the object
  - We write them as
    \[ \text{object . method ( additional, optional parameters )} \]
  - In fact, method is a function and object is the first and sometimes only parameter
Methods vs. Function

• There are two built-in ways to sort a list in Python:
  • The sorted function
  • The sort method for lists

• They are called differently because one is a method and one a function

  • sorted returns a sorted list
  • *.sort() does not return anything, but the list is sorted.

```
>>> lista = ['c', 'b', 'a', 'd']
>>> lista.sort()
>>> lista
['a', 'b', 'c', 'd']

>>> lista = ['c', 'b', 'a', 'd']
>>> sorted(lista)
['a', 'b', 'c', 'd']
```
Here is an overview of the most important list methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>append()</td>
<td>adds an element to the end of the list</td>
</tr>
<tr>
<td>clear()</td>
<td>removes all elements from a list</td>
</tr>
<tr>
<td>copy()</td>
<td>returns a copy of the list</td>
</tr>
<tr>
<td>count()</td>
<td>returns the number of elements in the list</td>
</tr>
<tr>
<td>extend()</td>
<td>adds the elements in the parameter to the list</td>
</tr>
<tr>
<td>index()</td>
<td>returns the index of the first occurrence of the parameter</td>
</tr>
<tr>
<td>insert()</td>
<td>inserts an element at the specified location</td>
</tr>
<tr>
<td>pop()</td>
<td>removes an element at the specified location or if left empty, removes the last element</td>
</tr>
<tr>
<td>remove()</td>
<td>removes the first element with that value</td>
</tr>
<tr>
<td>reverse()</td>
<td>reverses the order of the list</td>
</tr>
<tr>
<td>sort()</td>
<td>sorts the list</td>
</tr>
</tbody>
</table>
Range is not a list

- A range belongs to a data structure (called iterators) that are related to lists
  - In an iterator, you can always produce the next element
  - To make a list, just use the list keyword:

```python
lista = list(range(2, 1000))
```
Lists and for loops

- The for-loop in Python iterates through a list (or more generally an iterator)
  - for x in lista:
    - x takes on all values in lista
Checking membership

• In Python, membership in a list is checked with the `in` keyword
  • There is a more appealing, alternative form of negation

• Examples:
  • `if element in lista:`
  • `if element not in lista:`
    • Use this one instead of the negation around the statement
      • `if not element in lista:`
Sieve of Eratosthenes

- To calculate a list of all primes, we could:
  - Check all numbers in [2, 3, 4, … , n] that have no divisors
    - Which is tedious and does not scale to large n
  - Eliminate all multiples
    - This is the idea behind the famous Sieve of Eratosthenes
Sieve of Eratosthenes

• We start out with a list of all numbers between 2 and 1000
  • [2, 3, 4, 5, 6, 7, … , 999, 1000]
• The smallest number in the list is a prime, this would be 2
  • We can eliminate all true multiples of 2, that is, we remove 4, 6, 8, 10, … , 1000 from the list
  • This gives us
    • [2, 3, 5, 7, 9, 11, 13, …, 997, 999]
• The next smallest number has also to be a prime
Sieve of Eratosthenes

- [2, 3, 5, 7, 9, 11, 13, 15, 17, …, 997, 999]
- Therefore, 3, is a prime.
- For the next step, we eliminate all multiples of three that are left
  - [2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, … ,995, 997]
- We remove all multiples of 5 that remain in the list: 25, 35, 55, …
  - [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, … ,991, 997]
- And so we continue, until we can no longer eliminate multiples
Sieve of Eratosthenes

- We implement this in Python
  - We first define a function that removes multiples of an element from a list (of numbers)
  - We need one parameter \( \text{limit} \) to tell us when we should stop

```python
def remove_multiples(element, lista, limit):
    multiplier = 2
    while multiplier*element <= limit:
        if multiplier*element in lista:
            lista.remove(multiplier*element)
        multiplier += 1
```
Sieve of Eratosthenes

- We can now implement the sieve
  - We initialize a list to the first 1000 elements
  - We maintain an index to tell us to which of the elements we already processed

```python
def eratosthenes():
    lista = list(range(2, 1000))
    index = 0
```
Sieve of Eratosthenes

• We stop when the index is about to fall out of the current size of the list

• Don’t forget to increase the index

```python
def eratosthenes():
    lista = list(range(2, 1000))
    index = 0
    while index < len(lista):
        #Do the work here
        index += 1
```
Sieve of Eratosthenes

• The work to do for each index is to remove the multiples of the current element

```python
def eratosthenes():
    lista = list(range(2, 1000))
    index = 0
    while index < len(lista):
        element = lista[index]
        remove_multiples(element, lista, limit)
        index += 1
```
Sieve of Erathosthenes

• And here is the result, all primes until 1000

Sieve of Eratosthenes

- This implementation can be improved in a number of ways
  - For example, we do not need to remove all multiples because we know that they have been removed
    - For example, if we are processing 13, then we do not need to check for 2\times13, 3\times13, 4\times13, \ldots because they have already been replaced
  - And there are ways to implement it more elegantly, but the point is just to see how to program with lists.